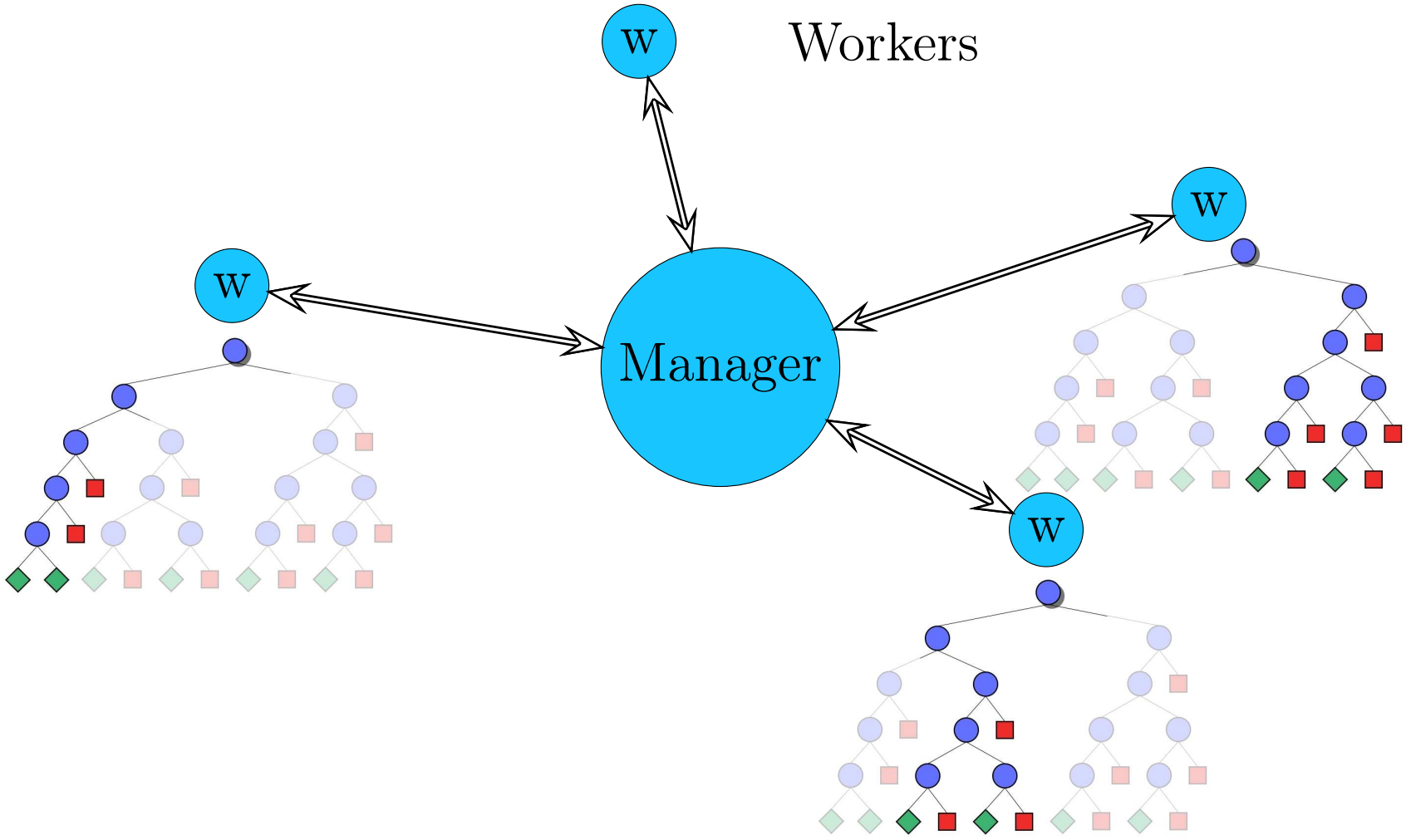
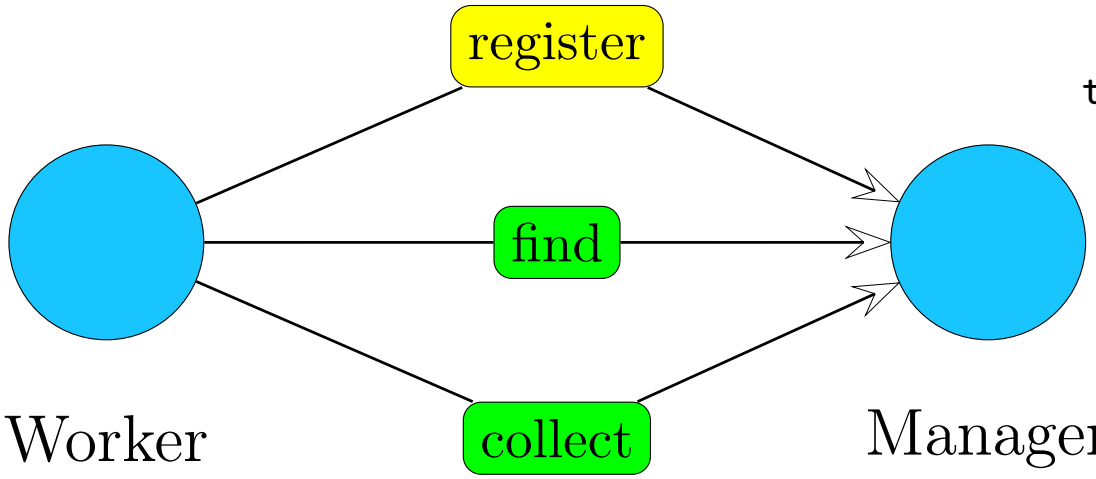
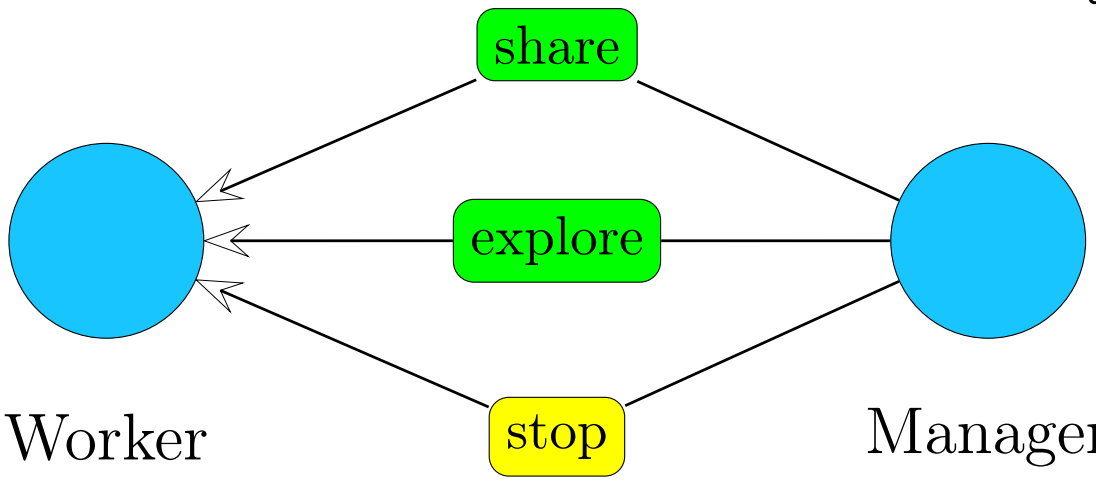
A distributed search engine



Interfaces




```
type α manager_intf =
    { register : α worker_intf → unit ,
      find     :           unit → unit ,
      collect  :              α → unit }
```




```
type α worker_intf =
    { share   :   unit → (α work) option,
      explore : α work → unit ,
      stop    :   unit → unit }
```

Manager Implementation

```
fun register workerIntf = ...
fun find     ()         = ...
fun collect  sol        = ...
```

Definitions

```
val remoteRegister = Remote.proxy register
val remoteFind     = Remote.proxy find
val remoteCollect  = Remote.proxy collect
```

Remote functions (RPC)

```
val managerInterface =
  {register = remoteRegister ,
   find     = remoteFind ,
   collect  = fn args => spawn (remoteCollect args)} (Asynchronous)
```

Export Interface

Distribution

```
val parcel = pack (val interface = managerInterface)
                :> (val interface : (int vector) manager_interface)
val url    = Remote.offer parcel
```

- `Remote.offer` starts a http server which "offers" the manager interface.
- We send the `url` by ssh

```
OS.Process.system "ssh -f host alicerun Worker url"
```

Workers

```
(* Get and unpack the parcel. *)
structure Parcel =
          unpack (Remote.take url)
            : (val interface : (int vector) manager_interface)

(* Get functions from the manager interface. *)
val managerIntf = Parcel.interface
val register    = #register managerIntf
val find        = #find managerIntf
val collect     = #collect managerIntf

(* Each function call automatically uses the remote implementation. *)
... find () ...
... collect solution ...
```

Conclusion

Distribution and concurrency are achieved using only a few well-defined primitives

- `Remote.proxy`, `Remote.offer` and `Remote.take`
- `pack` and `unpack`
- `spawn`

Static type-checking makes the use of these primitives safe with respect to types.